BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

Requested Patent: EP0940748A2

Title: OBJECT DISTRIBUTION IN A DYNAMIC PROGRAMMING ENVIRONMENT ;

Abstracted Patent: EP0940748 ;

Publication Date: 1999-09-08 ;

Inventor(s):

KING RICHARD ADAM (US); COHEN GEOFFREY ALEXANDER (US); KAMINSKY DAVID LOUIS (US) ;

Applicant(s): IBM (US) ;

Application Number: EP19990301529 19990302 ;

Priority Number(s): US19980036270 19980306 ;

IPC Classification: G06F9/46 ;

Equivalents: JP11312088

ABSTRACT:

A method and system are described which allow programs to become dynamically reconfigurable without programmer intervention. This means that the programs can be dynamically distributed among multiple computers within a computer network without modification to the source code of the programs running on the system. In addition, the method and system described allow an administrator of the system to specify conditions under which reconfiguration is to occur without modification to the source text of the program to be dynamically reconfigured.

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 940 748 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
08.09.1999 Bulletin 1999/36

(51) Int Cl.[6]: **G06F 9/46**

(21) Application number: **99301529.6**

(22) Date of filing: **02.03.1999**

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: **06.03.1998 US 36270**

(71) Applicant: **International Business Machines Corporation**
Armonk, NY 10504 (US)

(72) Inventors:
- **Cohen, Geoffrey Alexander**
  Durham, North Carolina 27705 (US)
- **Kaminsky, David Louis**
  North Carolina 27514 (US)
- **King, Richard Adam**
  Cary, North Carolina 27513 (US)

(74) Representative: **Waldner, Philip**
IBM United Kingdom Limited,
Intellectual Property Department,
Hursley Park
Winchester, Hampshire SO21 2JN (GB)

(54) Object distribution in a dynamic programming environment

(57) A method and system are described which allow programs to become dynamically reconfigurable without programmer intervention. This means that the programs can be dynamically distributed among multiple computers within a computer network without modification to the source code of the programs running on the system. In addition, the method and system described allow an administrator of the system to specify conditions under which reconfiguration is to occur without modification to the source text of the program to be dynamically reconfigured.

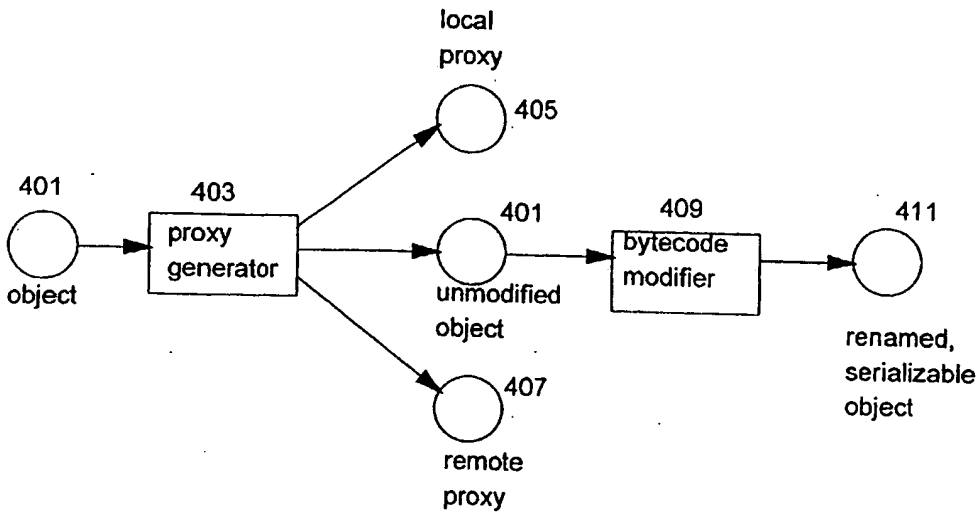


Fig. 4

EP 0 940 748 A2

Description

[0001] It has long been known that the ability to move objects dynamically from computer to computer is useful. For example, moving objects allows the system to load balance among computers. In addition, it allows the system to dynamically cluster objects that communicate frequently, reducing network traffic and improving overall system performance.

[0002] However, current systems that allow objects to be relocated (migrated) suffer significant drawbacks, typically requiring either a special purpose programming language (or special purpose modifications to a general purpose language) or a special purpose operating system. In the former case, the programmer must develop new skills and adorn their code with additional syntax to exploit object migration. In the latter case, the resulting programs are limited to execution on sparsely deployed systems.

[0003] The present invention, relating to Dynamic Object Distribution (DOD), describes a system that suffers neither of these limitations. In the preferred embodiment, the programmer simply writes his program in the Java programming language (Java is a trademark of Sun Microsystems) without adding any special syntax or notations. The program then executes on standard Java Virtual Machines (JVMs) which are widely deployed in the industry.

[0004] Object migration has been studied extensively in academia as well as in industry. Systems such as the v-kernel enable page-based migration on systems running the v-kernel, however, systems without the v-kernel cannot perform migration. Because the v-kernel requires operating system modification, it is not widely used. Systems such as Amber from the University of Washington migrate objects among nodes of a multi-processor computer. However, Amber requires the program to be written in a special language. The Amber runtime is also not widely used. A major benefit of the present invention is that it requires no kernel modifications and works in Java which is a widely-deployed language.

[0005] US Application Serial Number 08/852,263 entitled A Process for Running Objects Remotely filed on May 7, 1997 and assigned to the assignee of the present invention.

[0006] US Application Serial Number 08/941,815 entitled Apparatus and Method for Dynamically Modifying Class Files During Loading for Execution filed on September 30, 1997 and assigned to the assignee of the present invention.

[0007] According to one aspect of the present invention there is provided a method for dynamically distributing objects of a program, each object containing one or more programmed entities, from a first computer to one or more remote computers, said program having one or more objects, said method comprising the steps of:

identifying all of the objects in said program;
creating a list of conditions under which objects of said program are to be migrated;
for each object to be migratable, identifying each programmed entity to be accessed from outside of the object;
generating a first proxy and a second proxy for each object that may be moved to or accessed from a remote computer, said first proxy residing on the same physical device as said initial object and said second proxy created on said same physical device and transferred to said remote computer upon object migration, wherein said first proxy contains network linkage and indication to access said programmed entities on said remote computers and said second proxy contains network linkage and indication to access said programmed entities on said first computer; and,
moving an object from said first computer to a remote computer when said object s corresponding condition from said list of conditions is met.

[0008] According to a second aspect of the present invention there is provided computer readable code for dynamically distributing objects of a program, each object containing one or more programmed entities, from a first computer to one or more remote computers, said program having one or more objects, comprising:

first subprocesses for identifying all of the objects in said program;
second subprocesses for creating a list of conditions under which objects of said program are to be migrated;
subprocesses, responsive to said first and second subprocesses, for each object to be migratable, for identifying each programmed entity to be accessed from outside of the object;
subprocesses, responsive to said first and second subprocesses, for generating a first proxy and a second proxy for each object that may be moved to or accessed from a remote computer, said first proxy residing on the same physical device as said initial object and said second proxy created on said same physical device, and transferred to said remote computer upon object migration, wherein said first proxy contains network linkage and indication to access said programmed entities on said remote computers and said second proxy contains network linkage and indication to access said programmed entities on said first computer; and,
subsequent subprocesses for moving an object from said first computer to a remote computer when said object s corresponding condition from said list of conditions is met.

[0009] Dynamic Object Distribution (DOD), as defined by the present embodiment, takes as input:

- The set of Java class files comprising a software application. Each class file contains Java bytecodes (Java is a registered trademark of Sun Microsystems, Inc.).
- A list of objects that the programmer chooses to enable for migration. By default, DOD assumes that all objects are migratable.
- A list of the conditions ("predicates") under which DOD is to migrate the objects.

[0010] From the bytecode files, the DOD system generates local and remote "proxies" for the object. The proxies intercept method calls to and from the object, and route the calls to the object. When the object is local (it has not been migrated), the local proxy routes calls directly to the object; when the object is remote (it has been migrated), the local proxy routes calls to the remote proxy, which routes the calls to the object. A bytecode modification tool adjusts the names of the objects and references to the objects to ensure that name collisions do not occur between objects and their proxies.

[0011] As background, figure 1a illustrates calls from a method on an standard Java object to another standard Java object, in the illustration, calls from A **101** to B **105**. Figures 1b and 1c show a typical scenario of calls before and after an object has been migrated. Figure 1b shows calls being indirected through a local proxy **103** to the local object **105.** Figure 1c shows calls being sent from the local proxy **103** to a remote proxy **104**, then to the object itself **106** residing on the remote machine.

[0012] Based on the user-specified conditions (or predicates , as described above), a "migration thread" determines when an object should be migrated. It monitors the resources on the computer to determine when a predicate is satisfied. (Optionally, the migration thread can also gather information about remote computers. For example, it could gather such information using the standard Unix command 'ruptime,' which returns information about the processor load,of remote computers. Such information can be used for migration decisions.) To migrate the object, the migration thread instructs the local proxy to route method calls to the remote proxy, and to send a serialized version of the object and its remote proxy to the remote computer. At the remote computer, both objects are unserialized. Serialization is illustrated in Figure 7 and described further in Java in a Nutshell, second edition, O Reilly, May 1997, pp. 172-177. It is the flattening of an object so that it may be transmitted more efficiently through a network. An unserializer then reconstructs the original object at its destination.

[0013] It is an object of the present embodiment to provide a method for allowing programs to become dynamically reconfigurable without programmer intervention.

[0014] It is a further such object to allow a system to become dynamically reconfigurable without modification to the source text of the programs running on the system.

[0015] It is a further such object to allow the administrator to specify conditions under which reconfiguration is to occur without modification to the source text of the program to be dynamically reconfigured.

[0016] It is a further such object to allow the programmer to specify conditions under which the reconfiguration occurs on a per user basis.

[0017] It is a further such object to reconfigure the system dynamically and automatically without programmer intervention at the time of distribution.

Brief Description of the Drawings

[0018] Figure 1a illustrates calls between methods (prior art).

[0019] Figure 1b illustrates calls between methods in objects prior to migration (prior art).

[0020] Figure 1c illustrates calls between methods after migration (prior art).

[0021] Figure 2a depicts a standard object oriented (OO) program with three objects.

[0022] Figure 2b depicts a local configuration with one relocatable object.

[0023] Figure 2c depicts a migration thread for the configuration of figure 2b.

[0024] Figure 3 demonstrates operation after method C has been migrated to a remote host.

[0025] Figure 4 illustrates proxy generation and bytecode generation.

[0026] Figure 5 illustrates call return preparation.

[0027] Figure 6 exemplifies a migration thread.

[0028] Figure 7 depicts object serialization.

Detailed Description of the Preferred Embodiment

[0029] The present embodiment (DOD) uses a proxy scheme similar to Automatic Object Distribution (AOD, described in application serial number 08/852,263 referenced above), therefore, a brief review of AOD is presented herein.

AOD allows a programmer or system administrator to determine at any time before a program begins executing, how the program should be distributed onto a client and a server computer. AOD then automatically creates the code that allows the application to run as a distributed application.

[0030] However, as described in the AOD invention referenced above, AOD does not permit objects to be migrated between client and server at run-time. It requires the distribution to be complete prior to execution. It would be beneficial to move objects during runtime to adjust to varying conditions such as server load. That is, when the server becomes busier, more classes are moved to the client, reducing the server load. In the present invention an enhancement to AOD that allows objects to be moved during runtime is described.

[0031] First, the following example is used to review AOD. Consider an object 'a' instantiated from class A that contains (has a reference to) an object b instantiated from class B. B has a method foo(). Exemplary pseudocode for this situation is:

```
class A {
  A () {
    // constructor for A
  }
  some_method () {
    B b = new B (); // allocate object b
    b.foo(); // call foo method on b
  }
}
class B {
  B () {
    // constructor for B
  }
  foo () { // foo performs some action
  }
}
```

[0032] 'If the programmer determined that 'a' was to be split from b, the process would generate two proxies for B, B' and B". Calls from 'a' to 'b' would be intercepted by B', passed across the network to B" whiwh then makes a local call to B, passing the results back as necessary. Note that once this configuration has been established, it cannot be altered at run-time.

[0033] Dynamic Object Distribution (DOD) is an enhancement to AOD. In DOD, the programmer identifies not only which classes should initially be on the client, and which should initially be on the server, but also which classes might be moved dynamically from one to the other. This identification as to which classes might be moved can be made any time up until the program is run. The programmer identifies these classes not by changing the program itself (this is known art, see below), but by (e.g.) typing the list into a separate file. Other methods of specification will be apparent to those skilled in the art. At run time, programmer-specified "predicates" are used to trigger the automatic migration of the objects.

[0034] The DOD process is comprised of the following steps. These steps, as the preferred embodiment of current invention, are expressed in the Java programming language and execution environment, although other embodiments in other object-oriented programming systems are possible.

1) Writing the classes comprising an application in the Java programming language.
2) Compiling the source Java files into Java bytecode files.
3) Identifying which objects are available to be moved.
4) Specifying predicates (conditions) under which an object might move.
5) Specifying initial execution locations for objects instantiated from each class in the system.
6) Generating the local and remote migration proxies (described below).

[0035] The user then starts the program. As the program executes, the DOD process automatically moves objects when the predicates of step 4 are satisfied.

[0036] In addition to each of these steps, DOD contains a "migration thread." This thread is responsible for monitoring system resources to determine when the predicates (described in step 4, and further described below) have been satisfied. It also initiates object migration.

[0037] This process is now described in further detail. Steps 1 and 2 are standard parts of the application development for Java. In prior implementations, after step 2, the application would be executed by a user. However, that application would lack the ability to move objects dynamically.

[0038] Next, the classes that the developer desires to be mobile are identified. Objects instantiated from mobile classes are candidates for migration by DOD. If the programmer chooses not to enter any list, then DOD assumes that all objects can be migrated. Thus, by default, the programmer needn't make any source changes, nor need he do any additional work to enable object migration. This is a departure from known art. However, as we describe below, preparing an object for possible migration entails system overhead. The programmer can reduce this overhead by noting which objects will not be migrated. Still, requiring no changes to the program, even if a separate listing is required, provides significant benefit over the current art.

[0039] In step 4, the predicates are specified. These predicates are used to control the dynamic behavior of the system. The programmer identifies the conditions under which objects are migrated. For example, if objects designated O1 through O9 are to be executed on the server, the programmer can specify that DOD is to migrate O1 and O2 if the server load exceeds some threshold T1; O3, O4 and O5 are to be migrated if it exceeds T2; and O6 is to be migrated if it exceeds T3. O7, O8 and O9 are not migrated. In the preferred embodiment, this information is simply specified in a file called the "migration file," although other specification methods are clearly possible and would be obvious to one skilled in the art.

[0040] Note that migration files may be specified on a per user basis as well as on a more global basis. That is, in the preferred embodiment, each version of the migration file (alternatively, each section within a single migration file) is associated with one or more users of the application. Thus, the application can optionally exhibit different behavior when executed by different users.

[0041] Step 5 designates the initial configuration for the system. That is, the programmer tells the system which classes spawn server objects, and which spawn client objects. In the preferred embodiment, this information is also stored in the migration file.

[0042] An example of a migration file might be:

User1 MyClassOne CPU > .5 # move if CPU over half utilized
User1 MyClassTwo CPU > .7 # move if CPU over 70% utilized
User1 MyClassThree CPU > .9 # move if CPU over 90% utilized
User1 MyIOClassOne Disk < .4 # move if disk is 60% full
User1 MyIOClassTwo Disk < .2 # move if disk is 80% full ('#' indicates the beginning of a comment; subsequent text on the line is ignored.)

[0043] Generating the local and remote proxies is one key to the DOD system. This is illustrated by example. Returning to a modified version of the example shown above, assume that objects 'a' and 'b' are co-resident, and the programmer identified objects instantiated from B as migration candidates. DOD reads the bytecode for class B (of which 'b' is an instance) and determines all of its public methods. As in AOD, DOD generates a proxy for B called B'. All calls to B are then indirected through B'. In the case of local calls, the call sequence is A → B' → B. In the case of remote calls, as in AOD, the sequence is A -> B' -> B" -> B.

[0044] B' is then constructed to allow **both** local and remote calls. In exemplary pseudocode, B' looks like:

```
boolean local = TRUE; // object starts local
B () {  // constructor for the proxy for B
create a reference to the real B
}
foo () {
   if (local) make local call to the local foo();
  else make remote call to foo as described in AOD
}
}
```

[0045] As shown below, to each proxy, code must be added to migrate the object. In addition, code must be added that protects method calls against timing conflicts -- that is, calls against methods in the object are not permitted while the object is migrating. This is done by making the proxy class "synchronized," (a standard Java term) thus protecting the class against race conditions, and adding a migration method. Finally, the local proxy must register itself with the migration thread. This allows the migration thread to locate the proxy when an object must be migrated.

[0046] Each proxy is constructed such that it implements the Migratable interface. This allows the migration thread s register method to accept a single type of object as a parameter, that is, a proxy class implementing the Migration interfac . The migration thread s register method then stores references to the parameters (proxy objects) in a table for use when an object must be migrated.

[0047] Thus B' becomes:

```
synchronized class B implements Migratable {
  boolean local = TRUE; // object starts local
  B () { // constructor for the proxy for B
  create a reference to the real B
  register with the migration thread
  }
  migrate () {
    serialize the "real" b;
    send the serialized object via a socket to the partner where it
       will be instantiated
    local = FALSE;
  }
  foo () {
     if (local) make local call to the "real" foo();
    else make remote call to foo as described in AOD
  }
}
```

[0048] Note that since the proxy object B' is synchronized, by standard Java semantics, the migration method cannot be executed while another method in the object is being called; similarly, no method can be called while the object is being migrated.

[0049] Serialization of an object, and passing serialized objects over sockets is well-known art in Java. After the object is serialized and sent over the network (using a standard Java method called writeObject), it is received by a DOD component on the remote computer using another standard Java method (readObject). The unserialized object is now ready to execute on the remote computer.

[0050] Typically, to be serialized, classes must implement the Serializable interface provided in the standard Java release. However, instances of classes that do not implement the Serializable interface will not be serialized by the JVM. Yet, DOD requires that objects be migratable without source change, and Java requires that Serializable objects implement the Serializable interface.

[0051] This apparent problem is solved by identifying those classes that were listed by the user as mobile, but do not implement the Serializable interface. Using a known bytecode modification tool such as the JOIE tool described in the patent application designated above entitled Apparatus and Method for Dynamically Modifying Class Files During Loading for Execution, a transformation is provided that, at the point when the class is first loaded by the JVM, marks the class as implementing the Serializable interface. At this point, the execution can proceed as usual. Note that since the change -- forcing the class to implement the Serializable interface -- is done automatically by a bytecode modification tool, the programmer is not required to do anything, and no change to the source code occurs. Using a similar mechanism, parameter objects can be made to implement the Serializable interface if they must be transmitted from a local to a remote proxy. Figure 4 illustrates the process of proxy generation and bytecode modification.

[0052] In Figure 4, an object **401** is processed by a proxy generator **403**. The proxy generator creates a local proxy **405** and a remote proxy **407** as well as retaining the unmodified object **401**. The bytecode modifier **409** then processes the unmodified object **401** to create a serializable object **411** from it.

[0053] As in AOD, the remote proxy (B", in the example) accepts calls from the local proxy, and makes local calls to the original object (B). In addition, the remote proxy must include code to redirect calls from the remote object, through the remote proxy, through the local proxy, and finally to the callee.

[0054] As illustrated in Figure 5, the process of generating code for return calls is very similar to the process used in AOD and to the process used here for incoming calls. The bytecodes for the migratable object are inspected, and all method calls to other objects are extracted. For each such call, both a local proxy **510** and a remote proxy **520** are constructed containing similarly named methods to that discovered in the bytecodes. Code in the remote proxy ensures that when the method is called, a remote (RMI) call to the local proxy is executed. Code in the local proxy ensures that when it receives a remote call, it makes a local call to the actual callee. Thus, this process is nearly identical to the process for generating proxies for incoming calls, except that the object is inspected for outgoing calls instead of for potential incoming calls (that is, public methods). Note that normal functions of the local and remote proxies are reversed in this case.

[0055] Migrating the object back to the local machine simply entails reversing the process. The process is initiated by the migration thread when it detects that an "unmigration" should occur. The migration thread calls the "unmigrate" method on the local proxy.

[0056] It is important to ensure that timing problems do not occur during unmigration. Because the local proxy is synchronized, the call to the unmigrate method will not execute (that is, it will block) until a time when it will be the only method executing in the local proxy. Since all calls to the remote callee must pass through the local proxy, we know

that the remote callee will not be executing when the unmigration occurs. The unmigrate method on the local proxy simply calls the migrate method on the remote proxy. The call to the unmigrate method on the local proxy does not complete until the entire migration is complete.

[0057] The migrate method on the remote proxy performs the same function the migrate method on the local proxy performs. It serializes the object, transmits it to the local machine, which then unserializes the object. At that point, the local proxy sets its "local" boolean.

[0058] In the ongoing example, B' becomes:

```
synchronized class B {
  boolean local = TRUE; // object starts local
  B () {  // constructor for the proxy for B
   create a reference to the real B
   register with the migration thread
  }
  unmigrate () {
    make a remote call to migrate on the remote proxy
    receive the serialized object
    unserialize the object
    local = TRUE;
    return;
  }
  migrate () {
    serialize the "real" b;

    send the serialized object via a socket to the partner where it
       will be instantiated
    local = FALSE;
  }
  foo () {
     if (local) make local call to the "real" foo();
    else make remote call to foo as described in AOD
  }
}
```

[0059] The user then simply starts the system as they would any client/server Java application. As the application executes, DOD monitors the predicates specified in the migration file. When a predicate is satisfied (e.g., when a load threshold is satisfied), the migration process is triggered.

[0060] To perform the migration, when the migration thread of the DOD process detects that a predicate has been satisfied, the migration file is used to determine which objects are to be moved or migrated. They then call the migration method (shown above) on the proxies for each of these objects. The migration thread executes pseudocode illustrated in Figure 6.

[0061] The migration method on the local proxy serializes the object, it s proxies and the proxies of any objects which it calls and sends it to the remote node for execution. The JVM on the remote machine calls the "ReadObject" method to read the serialized objects, and registers the remote proxies with the Remote Method Invocation (RMI) Registry. Techniques for reading and writing serialized objects across a network via a TCP/IP socket and for interacting with the RMI Registry are well-known. In addition, the remote proxy (generated in step 6) for the migrated object is sent to the remote node.

[0062] The local proxy, as described above, is constructed such that it contains all of the public methods contained by the object which it is proxying. Each of these methods either calls a method on the actual object (if the actual object is local) or remotely calls (via RMI) a method on the remote object (if the object was migrated and is now remote).

[0063] To avoid changing the code for any object that calls the migratable object, the proxy takes the name of the object for which it is acting as a proxy. Thus, any calls originally destined for the migratable objects will be retargeted at the proxy simply by the standard Java semantics.

[0064] However, Java does not permit two objects to have the same name. In the case where the object is executing locally, both the proxy and the object being proxied will have the same name. That is impermissible. To rectify the problem, a renaming process is executed. Using a bytecode modification tool, the original (migratable object) is renamed by appending an arbitrary string to its name in its bytecode file. The constructors for the class are renamed similarly. The proxy then makes calls to this newly named object.

[0065] For example, consider the code fragment from above:

```
class A {
  A () {
    // constructor for A
  }
  some_method () {
    B b = new B (); // allocate object b
    b.foo(); // call foo method on b
  }
}
class B {
  B () {
    // constructor for B
  }
  foo () {
     // foo performs some action
  }
}
```

To create a proxy for B, first B is renamed in its bytecode ("class" file), then the proxy is created:

```
class Bwxyz { // this is the real, renamed B
  Bwxyz () {
    // constructor for the real class
  }
  foo () {
    // foo performs some action
  }
}
synchronized class B { // this is the proxy for B
  Bwxyz myBwxyz; // reference to the original object
  B () { // constructor for B
    myBwxyz = new Bwxyz (); // create a new object for the original B
    register with the migration thread
  }
  foo () { // calls foo on the renamed version of the original B
    myBwxyz.foo();
  }
  migrate () { // as above, object migration code, called by migr. thread
    serialize the "real" b;
    send the serialized object via a socket to the partner where it
       will be instantiated
    local = FALSE;
  }
  unmigrate () {
    make a remote call to migrate on the remote proxy
    receive the serialized object
    unserialize the object
    local = TRUE;
    return;
  }
}
```

[0066] The call from A to "B.foo" now refers to the "foo" method in the proxy object B. Thus, the B proxy has successfully intercepted calls to B. In turn, the B proxy makes calls through its reference to the original B.

[0067] Figure 2a shows a simple object oriented program comprised of 3 objects, A, B and C. Object A has three methods labeled a1-a3; B has four methods labeled b1-b4; and Object C has two methods labeled c1 and c2. As indicated in the figure, the following method calls exist:

a1 calls b3 and b4

a2 calls b2

b2 calls c2

c1 calls a2

[0068] Figure 2b depicts the same program after the programmer has specified that C is the only migratable object. The system automatically generated a local proxy specified in the figure as CL, containing two methods specified as

cl1 and cl2. Note that, per the discussion of naming conflicts, the local proxy and its methods are named with the original names of the object and its methods; the original object C was renamed by the bytecode modification tool (object renaming is not illustrated in the figure).

[0069] The method call list is thus adjusted to:

a1 calls b3 and b4
a2 calls b2
b2 calls cl2
cl2 calls c2
cl1 calls c1 (unused)
c1 calls a2

[0070] Since this is a local configuration, calls from the caller (e.g., B) are intercepted by the local proxy (CL) and are directed to the callee (C) via a standard method call.

[0071] Figure 2c shows the migration thread detecting a satisfied predicate and sending a message to the "migrate" method in the local proxy for C (CL). This causes the migration of C to the remote machine, and the transition to the configuration illustrated in figure 3.

[0072] Figure 3 shows the same configuration operating after C has been migrated to a remote computer. The method call list becomes:

a1 calls b3 and b4
a2 calls b2
// Call from b2 to c2 indirected remotely
b2 calls cl2
cl2 calls cr2 using RMI
cr2 calls c2
// Potential, but unused, calls into c1
cl1 calls cr1 using RMI (unused)
cr1 calls c1 (unused)
// Calls from c1 to a2
c1 calls ar2
ar2 calls al1 using RMI
al2 calls a2

[0073] As described above, DOD consists of the following main steps: the programmer writes and compiles an application. The result is a set of Java bytecode (class) files. This is a standard development phase for Java applications, and is not strictly a part of the present invention. Next the programmer identifies which objects can migrate, and the conditions under which they are to migrate. Then DOD generates the local and remote stubs. DOD performs bytecode modification to ensure that the appropriate classes implement the Serializable interface and that all name conflicts are resolved. The stubs contain the code to direct method calls to the object whether it is local or remote. It also contains the code necessary to migrate the object. Next the migration thread monitors the system resources, and determines when a predicate is satisfied. When a predicate is satisfied, it calls the migrate method on the object's proxy, causing a migration. Next the migration method serializes the remote object, along with the required remote proxies, and transmits them to the remote JVM (Java virtual machine), where the objects are reinstantiated. The application can then be restarted with its new configuration.

[0074] In summary a method and system are described which allow programs to become dynamically reconfigurable without programmer intervention. This means that the programs can be dynamically distributed among multiple computers within a computer network without modification to the source code of the programs running on the system. In addition, the method and system described allow an administrator of the system to specify conditions under which reconfiguration is to occur without modification to the source text of the program to be dynamically reconfigured.

[0075] As with the AOD referenced above, since this migration relies on breaks along method-call boundaries (which is standard convention and good programming practice), objects whose member variables are accessed directly cannot be migrated. To avoid difficulty, objects that are potentially migratable should contain only private member variables.

Claims

1. A method for dynamically distributing objects of a program, each object containing one or more programmed entities, from a first computer to one or more remote computers, said program having one or more objects, said method comprising the steps of:

   identifying all of the objects in said program;

creating a list of conditions under which objects of said program are to be migrated;
for each object to be migratable, identifying each programmed entity to be accessed from outside of the object;
generating a first proxy and a second proxy for each object that may be moved to or accessed from a remote computer, said first proxy residing on the same physical device as said initial object and said second proxy created on said same physical device and transferred to said remote computer upon object migration, wherein said first proxy contains network linkage and indication to access said programmed entities on said remote computers and said second proxy contains network linkage and indication to access said programmed entities on said first computer; and,
moving an object from said first computer to a remote computer when said object s corresponding condition from said list of conditions is met.

2. A method as claimed in claim 1 further comprising the steps of:

identifying which of all of the objects in the program are to be available for migration;
placing the names of the objects available for migration into a list; and,
utilizing said list as input to the distribution method.

3. A method as claimed in claims 1 or 2 wherein said moving of an object from said first computer to a remote computer comprises the steps of:

moving a copy of said object to said remote computer;
moving said second proxy to said remote computer; and,
directing said first proxy to call said second proxy on said remote computer.

4. Computer readable code for dynamically distributing objects of a program, each object containing one or more programmed entities, from a first computer to one or more remote computers, said program having one or more objects, comprising:

first subprocesses for identifying all of the objects in said program;
second subprocesses for creating a list of conditions under which objects of said program are to be migrated;
subprocesses, responsive to said first and second subprocesses, for each object to be migratable, for identifying each programmed entity to be accessed from outside of the object;
subprocesses, responsive to said first and second subprocesses, for generating a first proxy and a second proxy for each object that may be moved to or accessed from a remote computer, said first proxy residing on the same physical device as said initial object and said second proxy created on said same physical device and transferred to said remote computer upon object migration, wherein said first proxy contains network linkage and indication to access said programmed entities on said remote computers and said second proxy contains network linkage and indication to access said programmed entities on said first computer; and,
subsequent subprocesses for moving an object from said first computer to a remote computer when said object s corresponding condition from said list of conditions is met.

5. Computer readable code as described in claim 4 further comprising:

subprocesses, responsive to said first subprocesses, for identifying which of all of the objects in the program are to be available for migration;
subprocesses, for placing the names of the objects available for migration into a list; and,
subprocesses for utilizing said list as input to the distribution method.

6. Computer readable code as described in either claim 4 or claim 5 wherein said moving of an object from said first computer to a remote computer comprises:

subprocesses for moving a copy of said object to said remote computer;
subprocesses for moving said second proxy to said remote computer; and
subprocesses for directing said first proxy to call said second proxy on said remote computer.

7. A system contained within a computer network, said computer network having multiple computers connected together using telecommunications mechanisms, said system having a method for dynamically distributing objects of a program, each object containing one or more programmed entities, from a first computer to one or more remote

computers, said program having one or more objects, comprising:

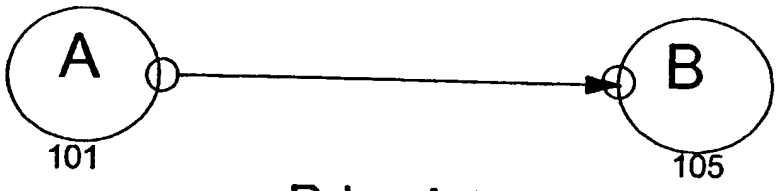
means for identifying all of the objects in said program;
means for creating a list of conditions under which objects of said program are to be migrated;
for each object to be migratable, means for identifying each programmed entity to be accessed from outside of the object;
means for generating a first proxy and a second proxy for each object that may be moved to or accessed from a remote computer, said first proxy residing on the same physical device as said initial object and said second proxy created on said same physical device and transferred to said remote computer upon object migration, wherein said first proxy contains network linkage and indication to access said programmed entities on said remote computers and said second proxy contains network linkage and indication to access said programmed entities on said first computer; and,
means for moving an object from said first computer to a remote computer when said object s corresponding condition from said list of conditions is met.

8. A system as described in claim 7 further comprising:

means for identifying which of all of the objects in the program are to be available for migration;
means for placing the names of the objects available for migration into a list; and,
means for utilizing said list as input to the distribution method.

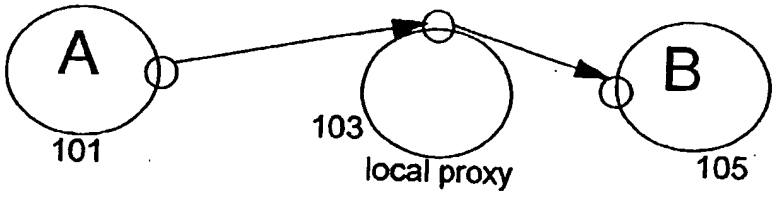
9. A system as described in either claim 7 or 8 wherein said moving of an object from said first computer to a remote computer comprises:

means for moving a copy of said object to said remote computer;
means for moving said second proxy to said remote computer; and,
means for directing said first proxy to call said second proxy on said remote computer.



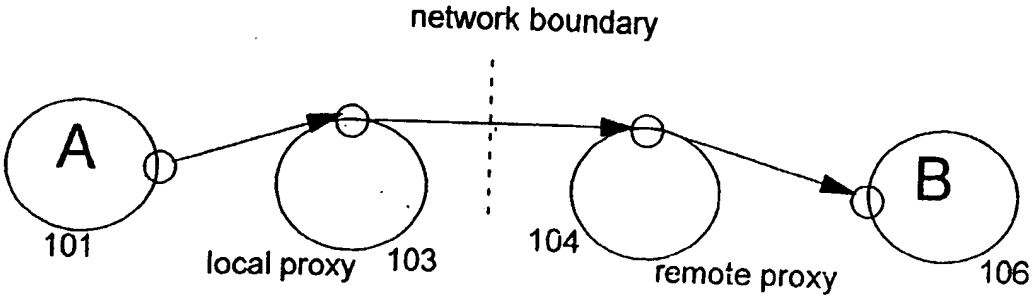
Prior Art

Fig. 1a



Prior Art

Fig. 1b



Prior Art

Fig. 1c

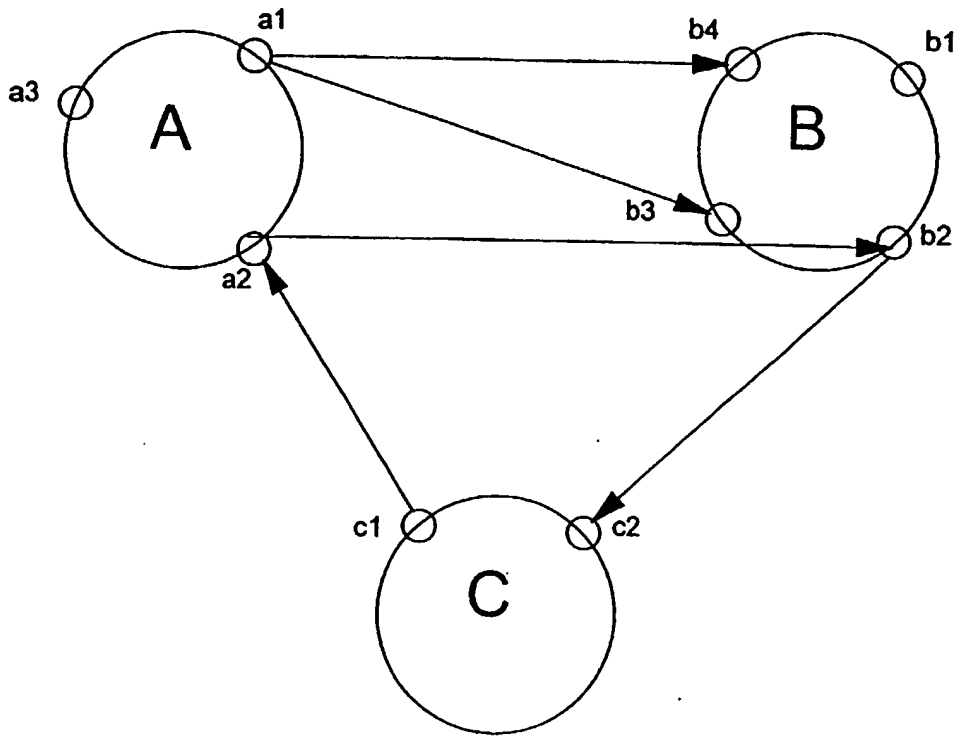


Fig. 2a

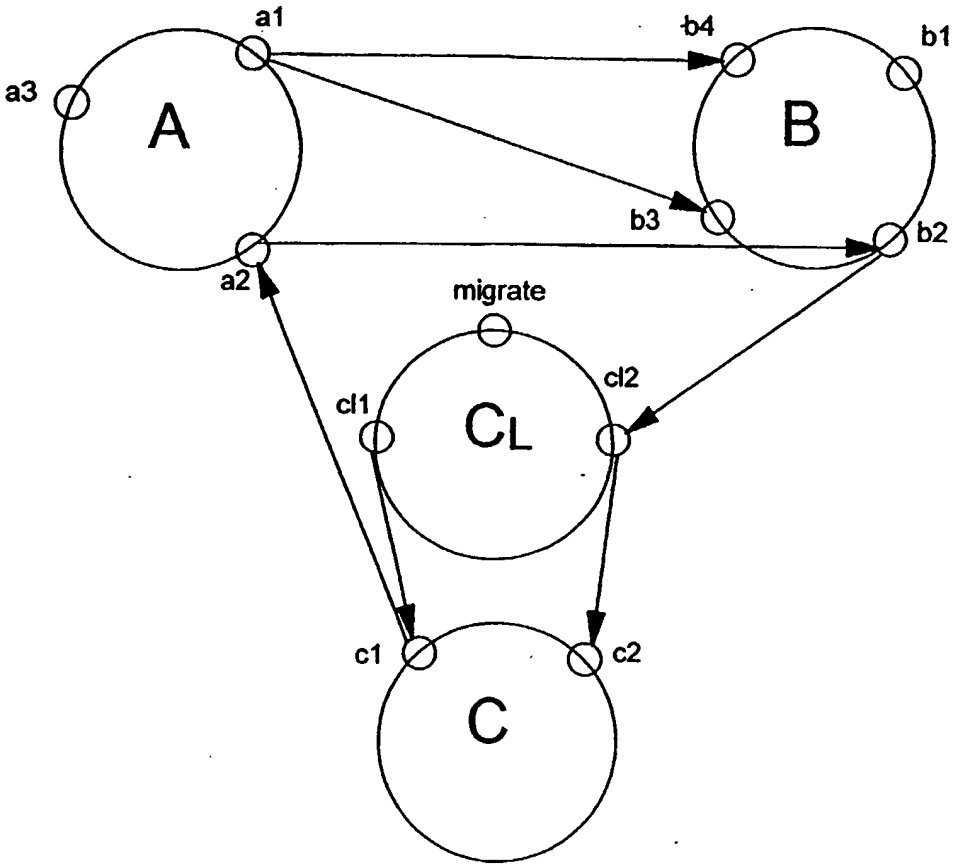


Fig. 2b

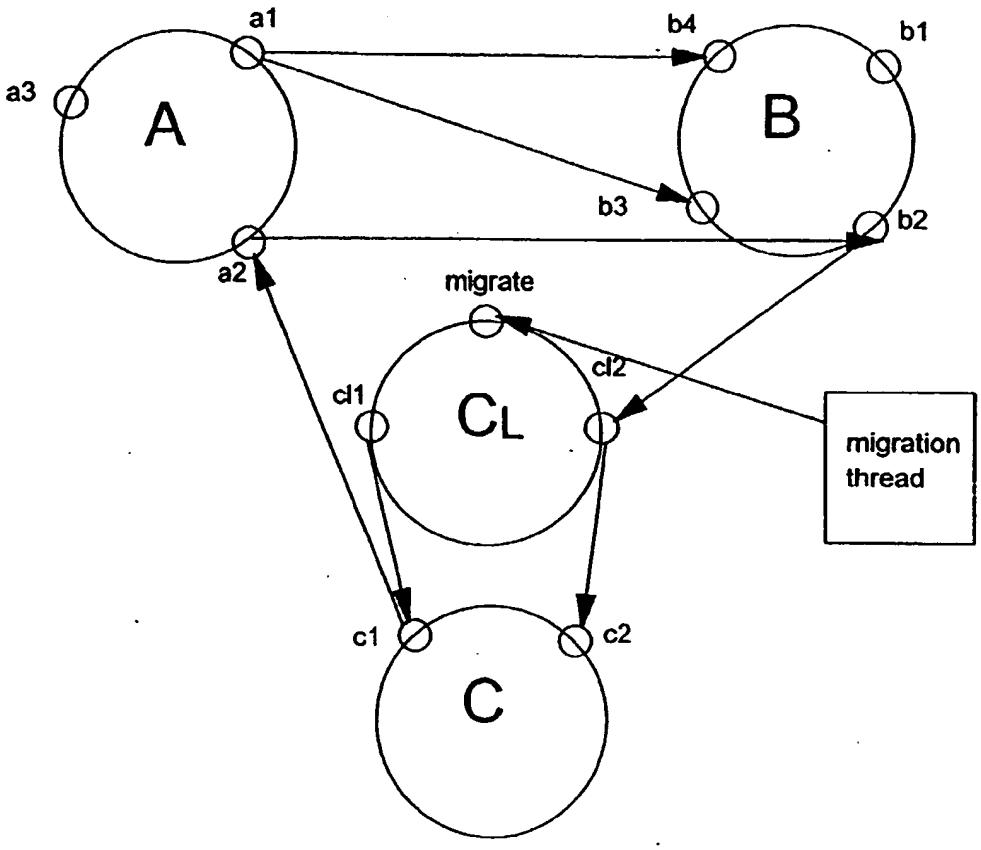

a1
b4
b1
a3
A
B
b3
b2
a2
migrate
cl2
cl1
CL
migration thread
c1
c2
C

Fig. 2c

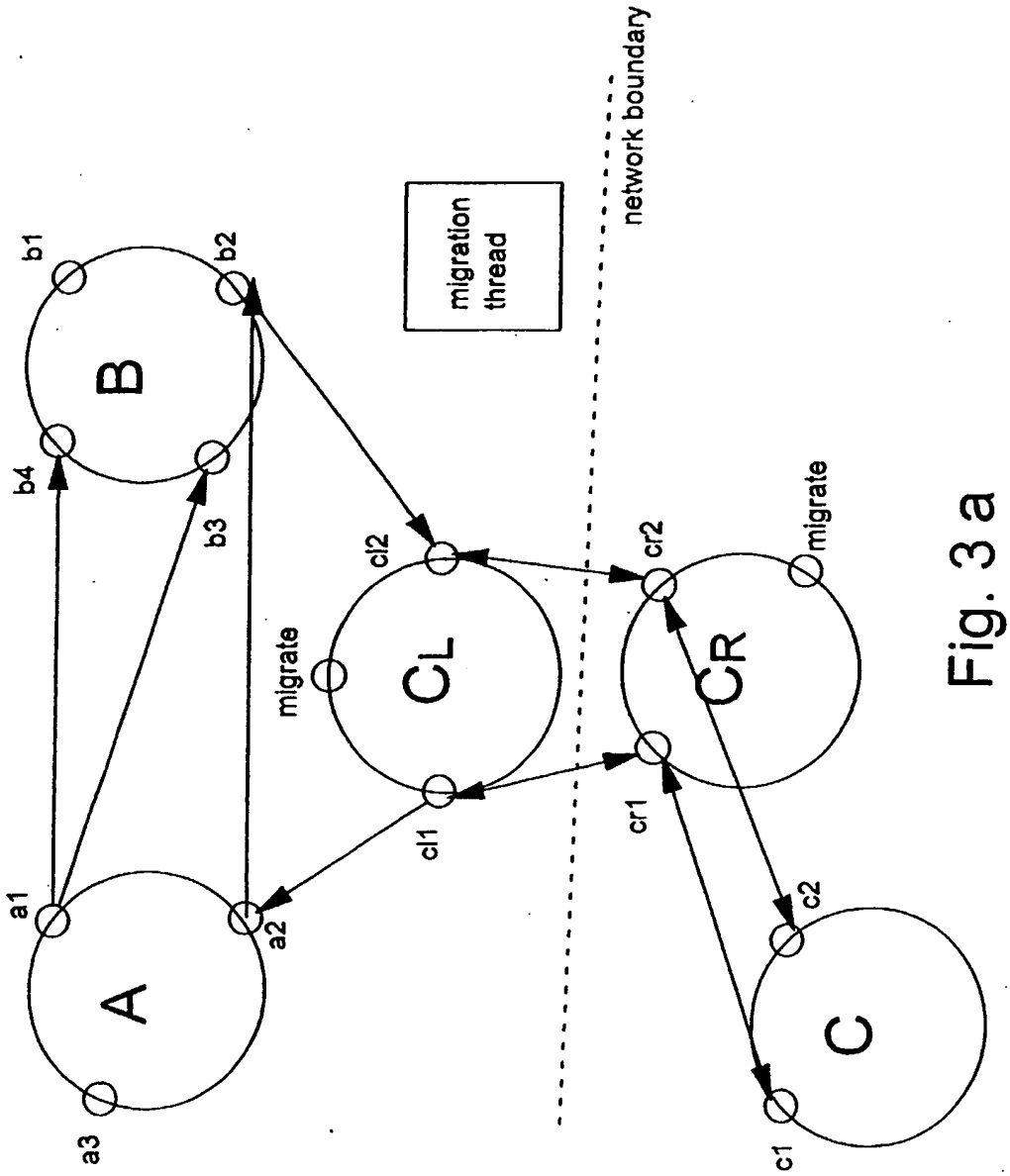

network boundary
migration thread
b1
b2
B
b4
b3
cl2
cr2
migrate
migrate
C_L
C_R
cl1
cr1
a1
a2
A
a3
c2
C
c1

Fig. 3 a

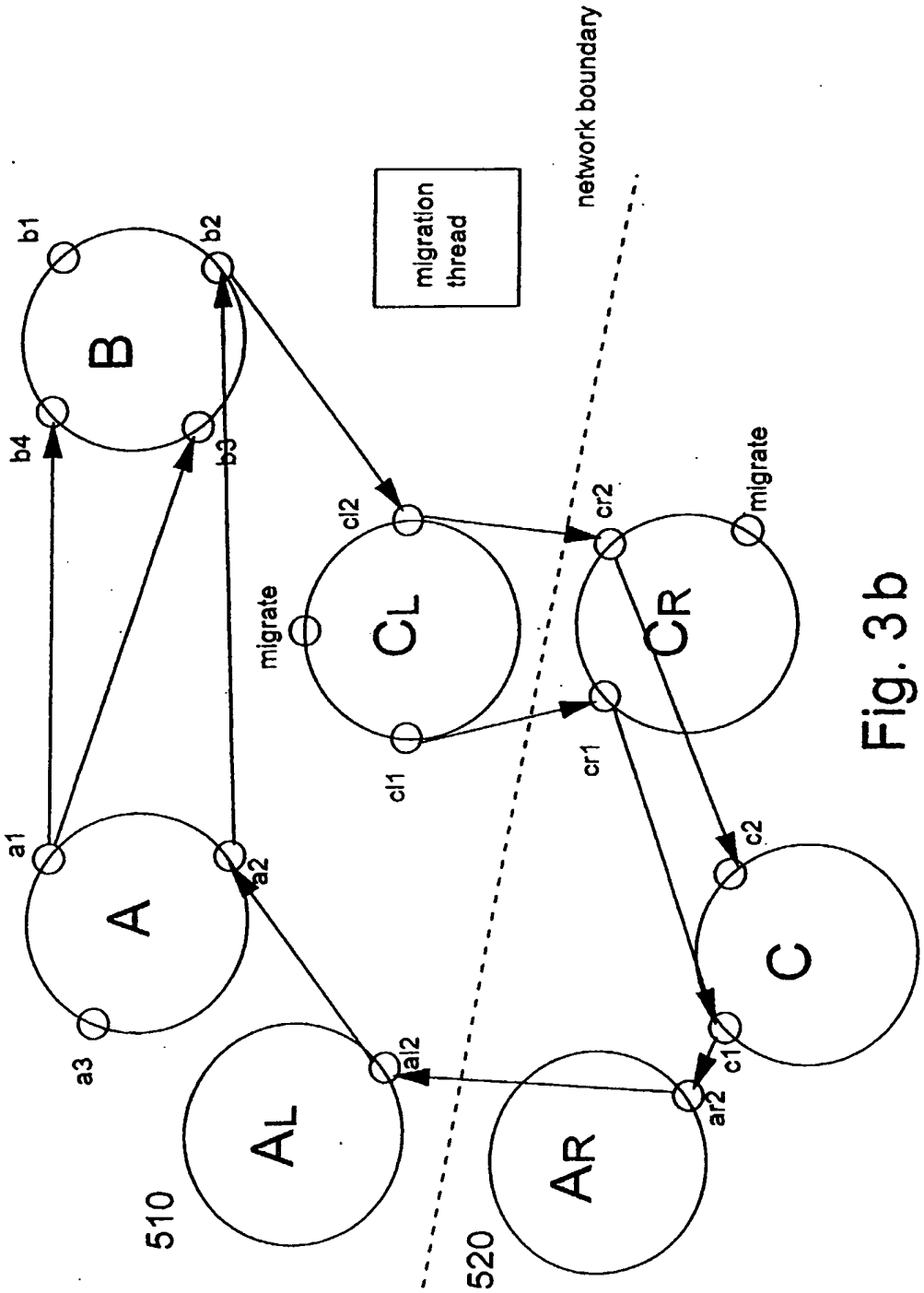

network boundary
migration thread
B
b1
b2
b3
b4
A
a1
a2
a3
migrate
CL
cl2
cl1
CR
cr2
cr1
migrate
C
c2
c1
AL
al2
510
AR
ar2
520

Fig. 3b

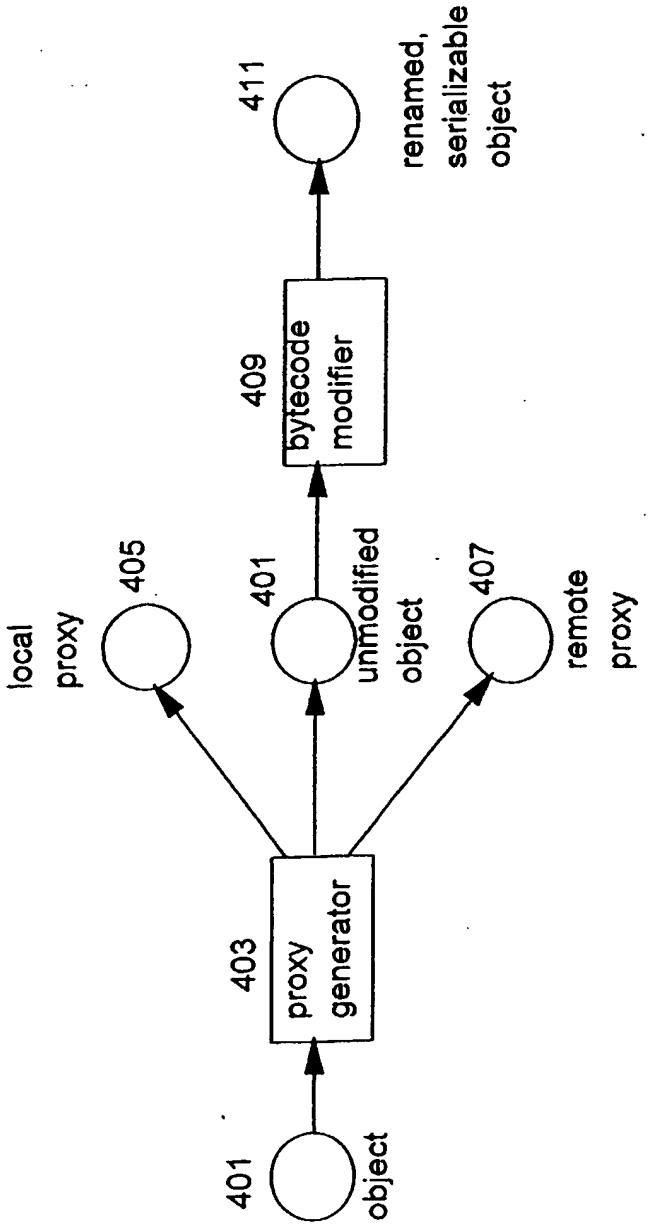

401
object
403
proxy
generator
local
proxy
405
401
unmodified
object
407
remote
proxy
409
bytecode
modifier
411
renamed,
serializable
object

Fig. 4

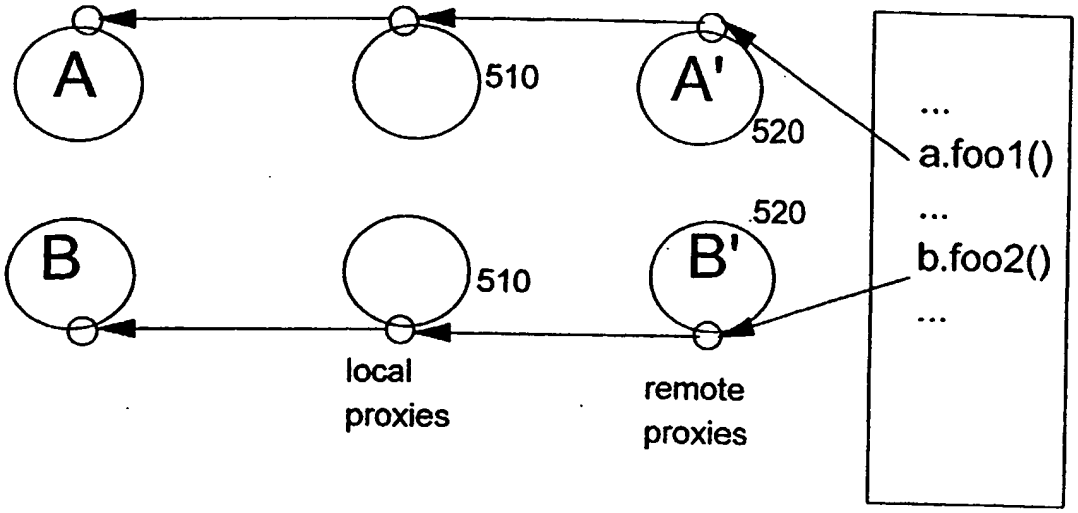

A
510
A'
520
...
a.foo1()
...
b.foo2()
...
520
B
510
B'
local
proxies
remote
proxies

Fig. 5

```
do forever {
  check for object registration
  check computer resource usage
  compare usages to predicates
  if a predicate is satisfied {
    check list of registered objects, initiating migration for
    those whose predicate was satisfied
  }
}
```

Fig. 6

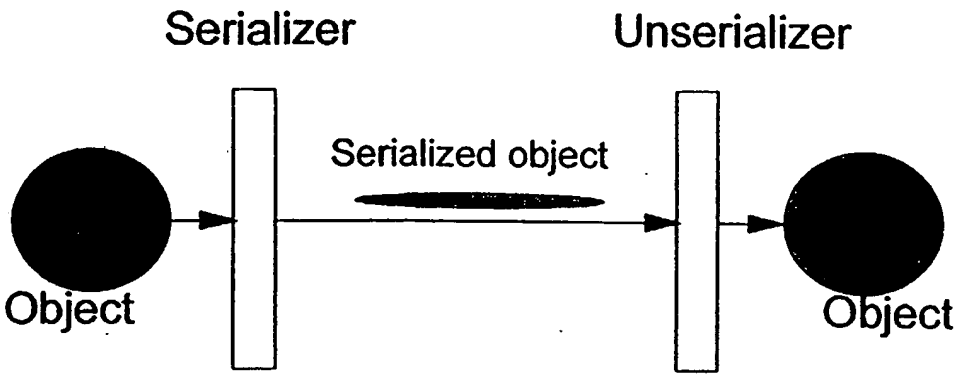

Serializer
Unserializer
Serialized object
Object
Object

Fig. 7